Report pf CM20219 Coursework

View/Analyse 3D Models using OpenGL

Table of Contents

Coursework 1 Part

Requirement 1: Draw a simple cube

Centred at (0,0,0), with opposite corner points (-1, -1, -1) and (1, 1, 1), faces orthogonal to x,y,z-axes.

Drawing a cube in OpengGl is basically drawing 6 cube faces, and drawing a face need 4 vertices with x,y,z coordinate. So, there is 8 points in total:
Example Code:

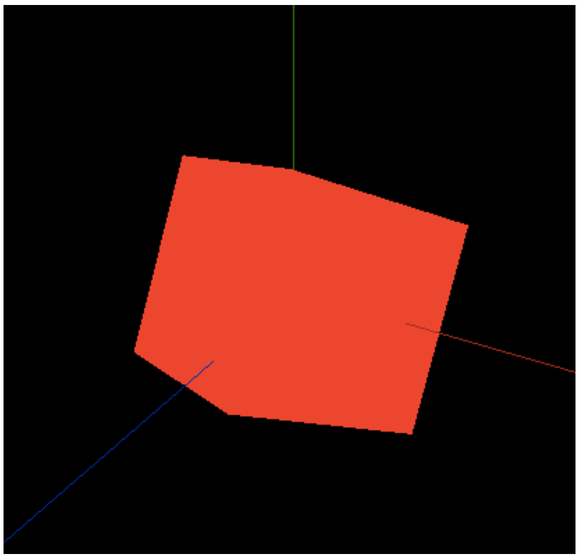
```
//Hold coordinates of all vertices
float vertex[8][3] ={{1,1,1},{-1,1,1},{-1,-1,1},{1,-1,1},{1,-1,-
1},{-1,-1,-1},{1,1,-1},{-1,1,-1}};
```

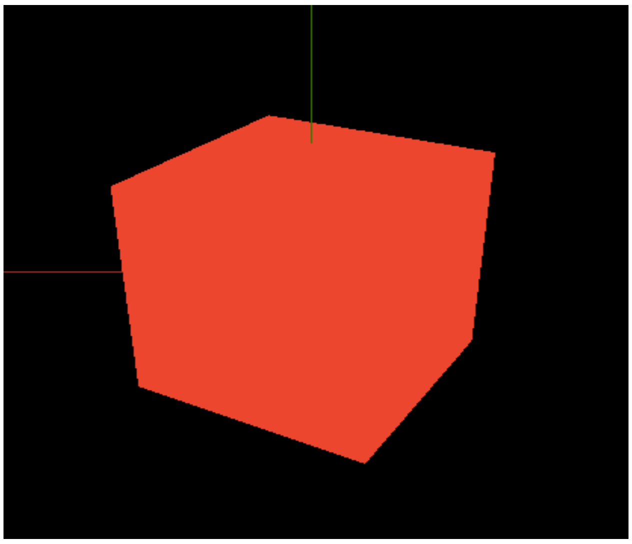
After locate the vertices of cube, creating an array that contains what vertex required to draw a specific face:
Example Code:

```
//Vertices for each face
int verticesForFace[6][4] =
{{0,1,2,3},{6,0,3,4},{6,7,5,4},{7,1,2,5},{6,7,1,0},{4,5,2,3}};
```

The above code means drawing the first face (verticesForFace[0]) needs the following vertex(verticesForFace[i]).
Thus, one face has been plotted. By using loop, we can draw the rest 5 faces:
Example Code:

```
/*
 Drawing cube and assign normal to each vertex.
 */
void drawCube(void)
{
    for (int i=0; i<6; i++) {
        glBegin(GL_POLYGON);
        glColor3f(1.0f,0.0f,0.0f); //Set color to red
        for (int j =0; j<4; j++) {
            glVertex3fv(vertex[verticesForFace[i][j]]);
        }
        glEnd();
    }
}
Screenshot for the result:
```





Requirement 2: Show coordinate system axes as lines

Draw lines to represent the axes – no arrows or tick marks necessary

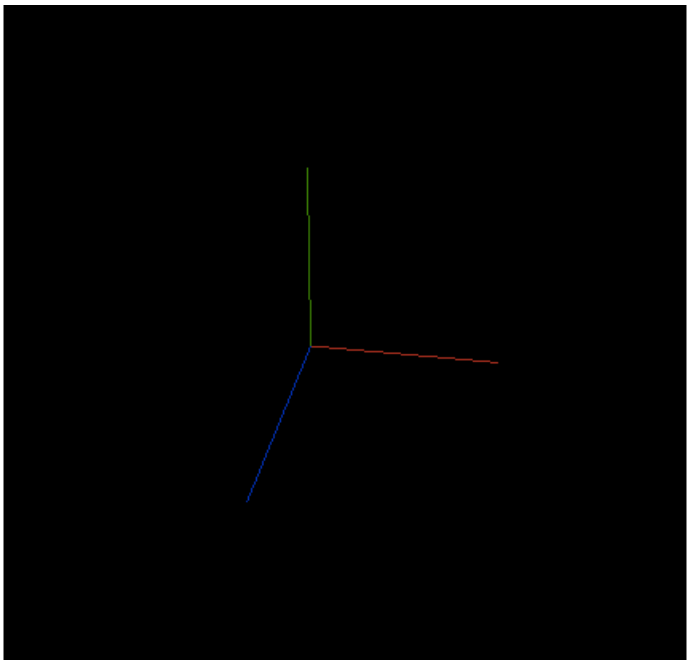
Drawing axes be implanted by providing two points which lies on each axes, and draw a line across them.

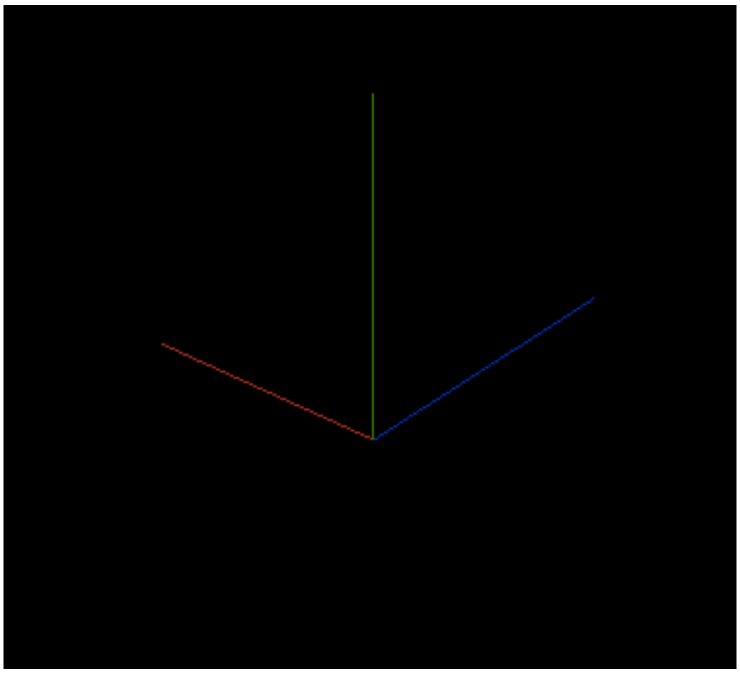
Example Code:

```
void drawAxis(){
    glBegin(GL_LINES);
    //X AXIS
    glColor3f(0.7, 0, 0);
    glVertex3f(0, 0, 0);
    glVertex3f(3, 0, 0);
    //Y AXIS
    glColor3f(0, 0.5, 0);
    glVertex3f(0, 0, 0);
    glVertex3f(0, 3, 0);
    //Z AXIS
    glColor3f(0, 0, 0.7);
    glVertex3f(0, 0, 0);
    glVertex3f(0, 0, 4);
    glEnd();
}
```

`The above codes draw threes lines start at (0,0,0) and along x,y,z axis. Color for each axes: Red for X, Green for Y, Blue for Z.`

Screenshot for the result:





Requirement 3: Rotate, zoom, and pan the camera

by using gluLookAt (axes should move with the cube)

gluLookAt takes 3 kinds of arguments : the camera position, the target position and the up vector. In fact, to build a look at matrix, we need to know 3 vectors that is perpendicular to each other representing 3 axes, and 1 positon vector. According to this, we need Right vector, Up vector, Direction vector and camera positon. The first three vectors create a space, combine this with a translation vector we can create a look at matrix, we can multiply this look at matrix by any vector to transfer to reference frame. This is our look at matrix:

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And in OpenGL, the gluLookAt help us doing this multiplication, so all we need is to compute the vector and pass them to gluLookAt.

Rotating the camera

The first three parameters of gluLookAt (eyeX,eyeY,eyeZ) represent the posting of camera, so performing camera rotation can be implemented by working out the resultant position (eyeX' eyeY' eyeZ') .

Example Code:

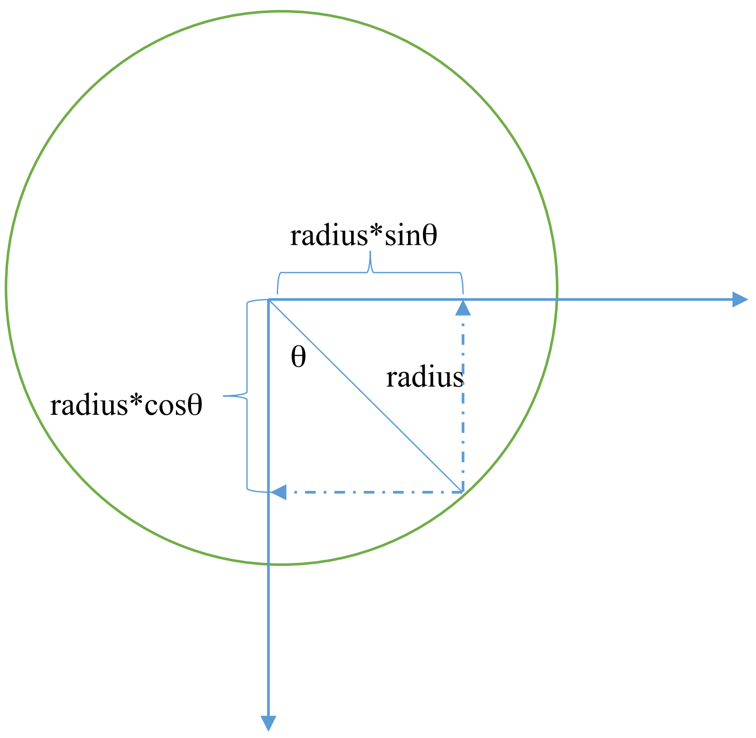
```
eyez = radius*cos(theta*M_PI/180)*cos(alpha*M_PI/180);
eyey = radius*-sin(alpha*M_PI/180);
eyex = radius*-sin(theta*M_PI/180)*cos(alpha*M_PI/180);
```

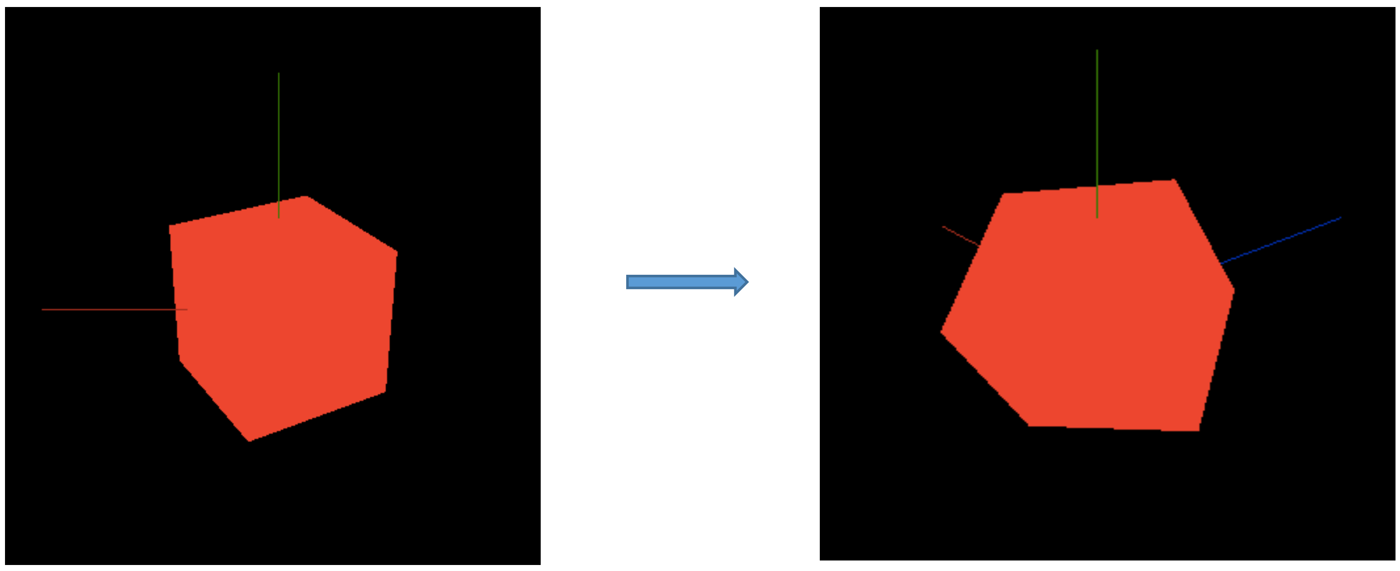
Considering a pointing orbiting around the origin, it requires angle (float theta,alpha), radius (double radius) and the resultant point position (double eyeX, eyeY,eyeZ).

As the figure shows, in XZ plane, radius *cosθ = eyeY, radius*sinθ = eyeX, and eyeY does not change. Similarity, we can obtain the eyeY and eyeZ in YZ plane: eyeY = radius*sinα, eyeZ = radius *cosα, and eyeY,eyeZ in XZ plane : eyeY = radius*sinα, eyeZ = cosα.

By working out the product of two points of eyeZ and eyeX, we can get the resultant points in 3D space.

After knowing how to calculate the resultant points p', we can perform rotation, by increasing/decreasing the degree of theta (along XZ plane) and alpha (along YZ,YX plane).



Finally, assigning a key callback to modify the value of angle to keep camera rotating.
Example Code of camera rotation about Y axis by 10 degrees:
```
case '[':    //Key press
            theta-=10;//Set rotation angle
            eyez = radius*cos(theta*M_PI/180)*cos(alpha*M_PI/180);
            eyey = radius*-sin(alpha*M_PI/180);
            eyex = radius*-sin(theta*M_PI/180)*cos(alpha*M_PI/180);
            cameraRotation =1;
break;
Configuration of gluLookAt:
gluLookAt(eyex,eyey ,eyez ,
          0.0f, 0.0f, 0.0f,
          0.0f, 1.0f, 0.0f);
Screenshot for the result:
```


Zooming the Camera

Zooming in and out the camera can be achieved by changing the FoV in `gluPerspective`.
Example Code:

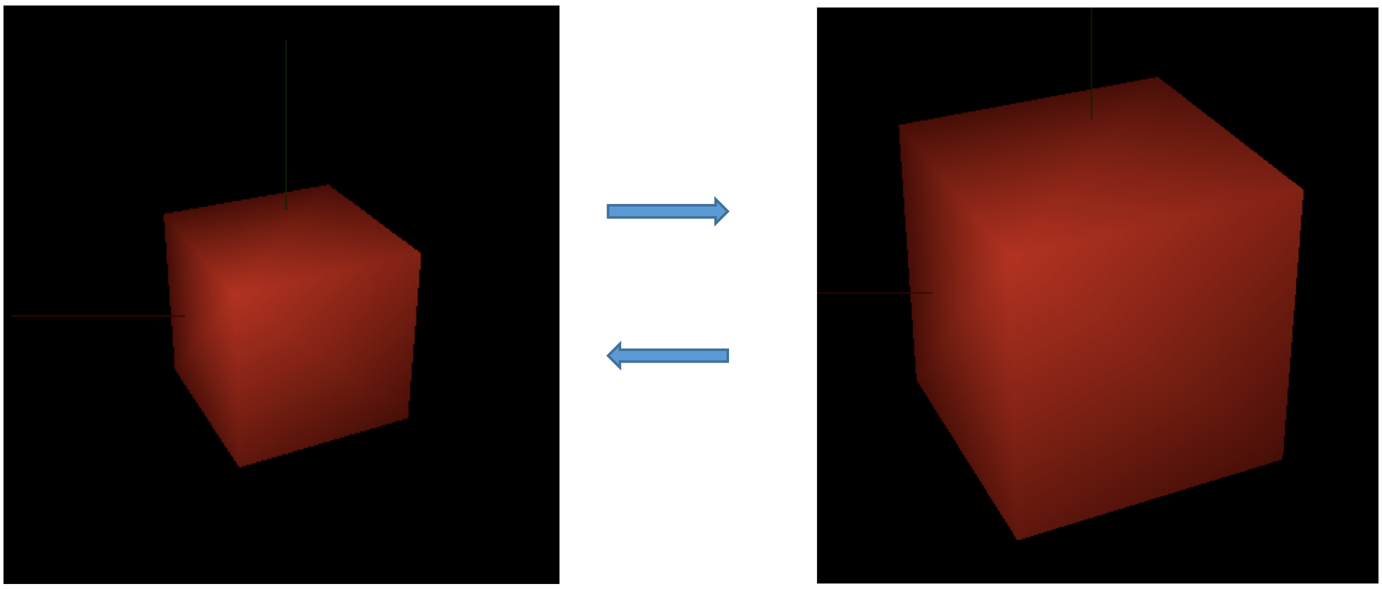
```
gluPerspective(fov,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
```

By assigning a callback callback function to increase or decrease the value of FoV, zooming the camera is implemented.
Example Code:

```
/*
 change the value of fov field of view by value of Y.
 */
void mouseMove(int x, int y)
{
    if (mousePress)
    {
        fov = y/10;
        if (fov <= 1.0f)
            fov = 1.0f;

        if (fov >=45.0f) {
            fov = 45.0f;
        }
    }
    glutPostRedisplay();

}
```

In the above code, I change the FoV by moving the mouse forward and backward according to the difference between value of Y.

Screenshot of the result:



Panning the camera

While panning the camera, we also changed the the target position which is the second parameters, so the target positon cameraFront = cameraPosition + cameraFront. To strafe camera to the right, we also need a direction vector pointing to the right the Right Vector. To obtain the Right Vector, we need the cross product of up vector and cameraFront and normalise it. By multiplying a value to the right vector and assign the result to camera position, the camera can now panning along the X axis.

Example Code:

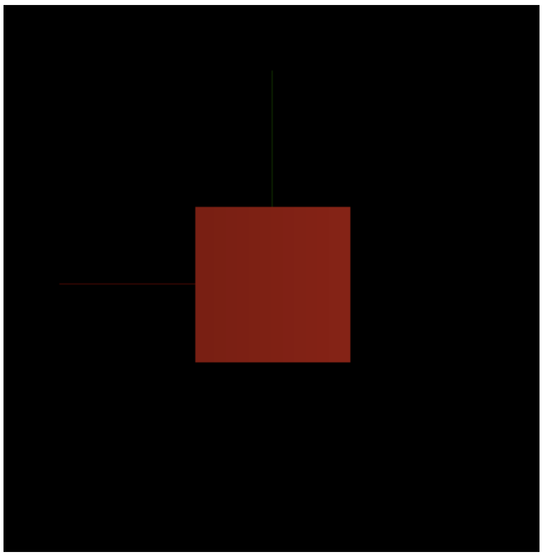
```
case 'D':   // Key press
float *temp2;
temp2 = calCrossProduct(cameraFront,  cameraUp);  // Right vector
cameraPos[0] += moveSpeed*temp2[0];
cameraPos[1] += moveSpeed*temp2[1];
cameraPos[2] += moveSpeed*temp2[2];
break;
```

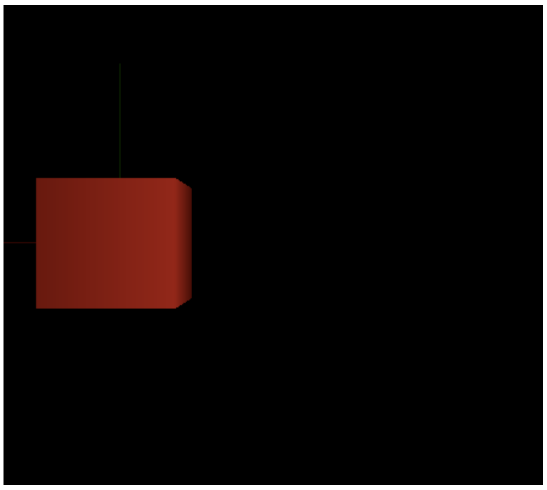
Similarily. we can panning camera on the opposite side by subtraction:

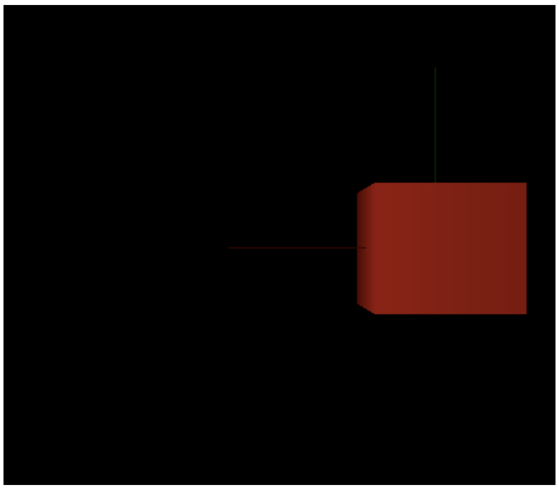
```
case 'A':   // Key press
float *temp;
temp = calCrossProduct(cameraFront,  cameraUp); // Right vector
cameraPos[0] -= moveSpeed*temp[0];     // subtract the value
cameraPos[1] -= moveSpeed*temp[1];     // to move left
cameraPos[2] -= moveSpeed*temp[2];
break;
```

```
Configuration of gluLookAt:
float *target;
    target= vectorAddition(cameraPos, cameraFront);
    //set normal camera
    gluLookAt(cameraPos[0], cameraPos[1], cameraPos[2],
              target[0], target[1], target[2],
              0.0f, 1.0f, 0.0f);
```

Screenshot of the result:







Requirement 4: Rotate cube

about x, y, z-axes respectively (axes should not move)

Rotation can be implemented by given three Euler Angles (θx θy θz) and forming the following rotation matrix Rx,Ry,Rz :

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{bmatrix} \quad Y = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix} \quad Z = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Example Code:

```
//RotationX matrix.
float Rx[4][4] = {{1,0,0,0},
                 {0, cosf(-meshAngleX),-sinf(-meshAngleX),0},
                 {0,sinf(-meshAngleX),cosf(-meshAngleX),0}};
//RotationY matrix
float Ry[4][4] = {{cosf(-meshAngleY),0,sinf(-meshAngleY),0},
                 {0,1,0,0},
                 {-sinf(-meshAngleY),0,cosf(-meshAngleY),0}};
//RotationZ matrix
float Rz[4][4] = {{cosf(-meshAngleZ),-sinf(-meshAngleZ),0,0},
                 {sinf(-meshAngleZ),cosf(-meshAngleZ),0,0},
                 {0,0,1,0}};
```

After having these matrices, applying them to each vertex in the cube to work out the resultant coordinates after each rotation.
Example code for rotating cube along X axis:

```
void rotateCubeX(){
//Applying Rx matrix to every points in cube vertices.
for (int i=0; i<vertices.size(); i++) {
    float x = vertices[i][0];
    float y = vertices[i][1];
    float z = vertices[i][2];
        vertices[i][0] = (Rx[0][0] * x) + (Rx[0][1] * y) + (Rx[0][2]
* z) + Rx[0][3];
        vertices[i][1] = (Rx[1][0] * x) + (Rx[1][1] * y) + (Rx[1][2]
* z) + Rx[1][3];
        vertices[i][2] = (Rx[2][0] * x) + (Rx[2][1] * y) + (Rx[2][2]
* z) + Rx[2][3];
    }
}
```

Finally, assigning a key callback to call these rotation function:
Example Code:

```
// Cube operation
  case 's':
  rotateCubeX();
  break;
```

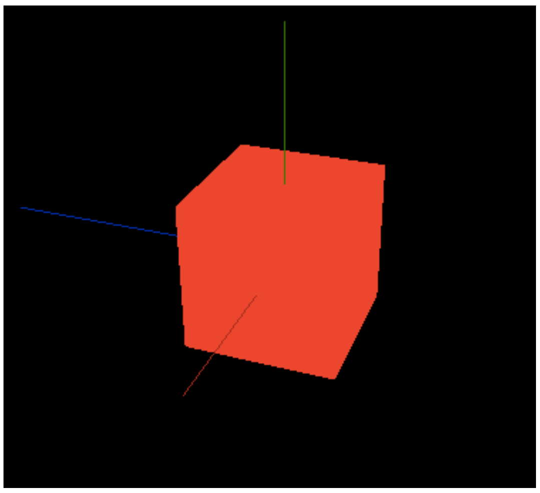
Another simpler approach towards this requirement is by using OpenGL function glRotate.
Example Code for cube rotation:

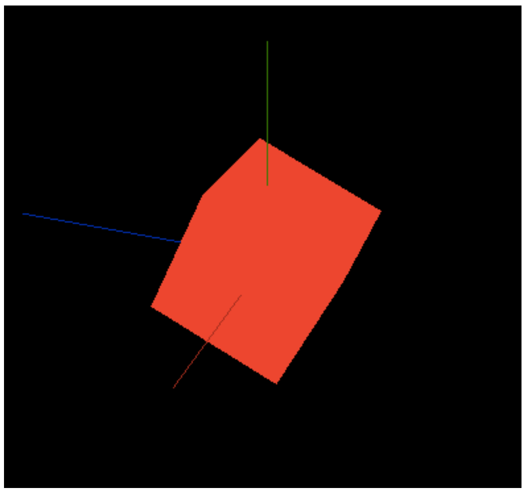
```
// Rotation
    glRotatef(rotateX, 1.0, 0.0, 0.0);
    glRotatef(rotateY, 0.0, 1.0, 0.0);
    glRotatef(rotateZ, 0.0, 0.0, 1.0);
```

By adjusting the value of rotate X,Y,Z , the cube can rotate along X Y Z axes respectively.
Example Code of modifying rotateX value by key:

```
//Cube operation
  case 's':
  rotateX -= 5;
  break;
  case 'w':
  rotateX += 5;
  break;
```

Screenshot for the result:





Requirement 5: Different render mode

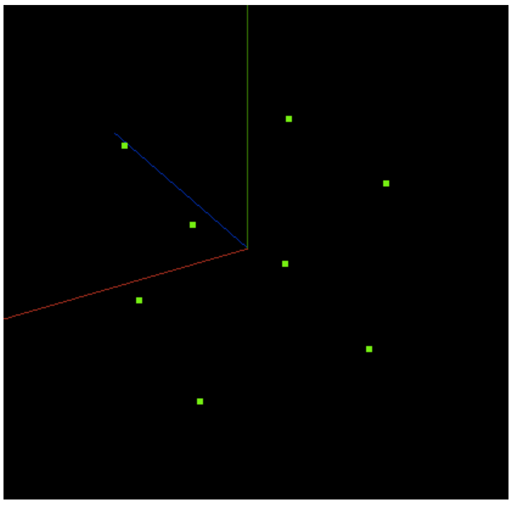
show only vertices/edges/faces – manually add light source in OpenGL and specify cube material to create shading effect (Phong illumination)

Three different render modes

Showing faces has been implemented by creating cube using GL_POLYGON from Req 1. Idea of showing vertices and edges are similar to drawing a cube, drawing vertices just replace GL_POLYGON to GL_POINTS.
Example Code of drawing vertices of the cube:

```
/*
 Drawing cube with vertices only
 */
void drawVertex(){
    for (int i=0; i<6; i++) {
        glBegin(GL_POINTS);
        glColor3f(0.0f,1.0f,0.0f);    //Color blue

        for (int j=0; j<4; j++) {

            glVertex3fv(vertex[verticesForFace[i][j]]);

        }
        glEnd();
        glPointSize(5);
    }
}
```

Screenshot of the result:



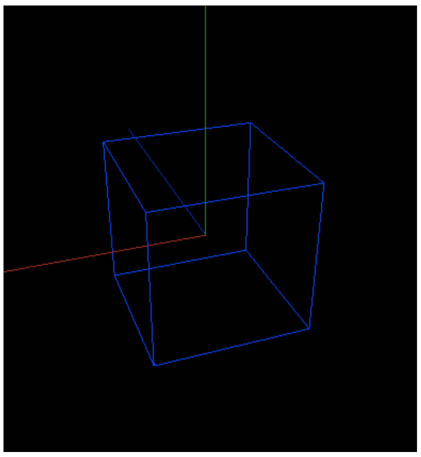
In terms of edges, the idea is drawing lines (GL_LINES) between two vertices. According to this, one face has 4 lines and need 6 points to achieve this. Therefore, we need a 2D-array to store the vertex number required to draw lines in each face.
Example code:

```
int verticesForFace[6][4] =
{{0,1,2,3},{6,0,3,4},{6,7,5,4},{7,1,2,5},{6,7,1,0},{4,5,2,3}};
```

So, face one(front) need the vertex in vertex[8][3] 0,1,2,3 index position to draw its edges.
Example code:

```
/*
```

```
 Drawing cube with edges only.
 */
void drawEdges(){
    for (int i=0; i<6; i++) {
        glBegin(GL_LINES);
        glColor3f(0.0f,0.0f,1.0f);
        for (int j=0; j<8; j++) {
            glVertex3fv(vertex[edgesMap[i][j]]);
        }
        glEnd();
    }
}
Screenshot of the result:
```



Assigning key callback function to manually show the above cases.
Example Code of key assignement:

```
void display (void) {
    . . .
    // different render mode*/
    switch ( rendermode ) {
        case 'f': // to display faces
            drawCube();
            break;
        case 'v': // to display points
            drawVertex();
            break;
        case 'e': // to display edges
            drawEdges();
            break;
    }
    . . .
}
void keyboard ( unsigned char key, int x, int y )
{   . . .
     // Switch render mode for v,e,f
     case 'v':
         rendermode='v';
         break;
     case 'e':
         rendermode='e';
         break;

     case 'f':
         rendermode='f';
         break;
    . . .
}
```
So we can render the cube in three different modes: faces, vertices, edges.

Creating Shading effect

First of all, we need a light source, in OpenGL, adding a light source need first specify the parameters of the light source in order to create Phong Illumination: diffuse, ambient, specular and light position:

```
    GLfloat light_position[4] = { 1.0, 0.0, 1.0, 1.0 };
    GLfloat light_amb[4] = {0.1,0.1,0.1,1.0};
    GLfloat light_diff[4] = {0.5,0.5,0.5,1.0};
    GLfloat light_spec[4] = {0.2,0.8,0.2,1.0};
And pass these parameters:
    glLightfv( GL_LIGHT0, GL_POSITION, light_position );
    glLightfv( GL_LIGHT0, GL_AMBIENT, light_amb );
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diff);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_spec);
Finally, enable it:
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
```

The material of cube also affect the result of illumination, thus, we need to specify the material of the cube:

```
    glMaterialfv(GL_FRONT, GL_AMBIENT, refl_amb);
    glMaterialfv(GL_FRONT, GL_SHININESS, refl_shininess);
    glMaterialfv(GL_FRONT, GL_DIFFUSE,refl_diff);
    glMaterialfv(GL_FRONT, GL_SPECULAR,refl_spec);
```

More than that, we also need the normal for each vertex. A normal of a face can be obtained by calculating cross products of two vectors which lie on the same face and normalise them.

Example code of a function that calculate the cross product:

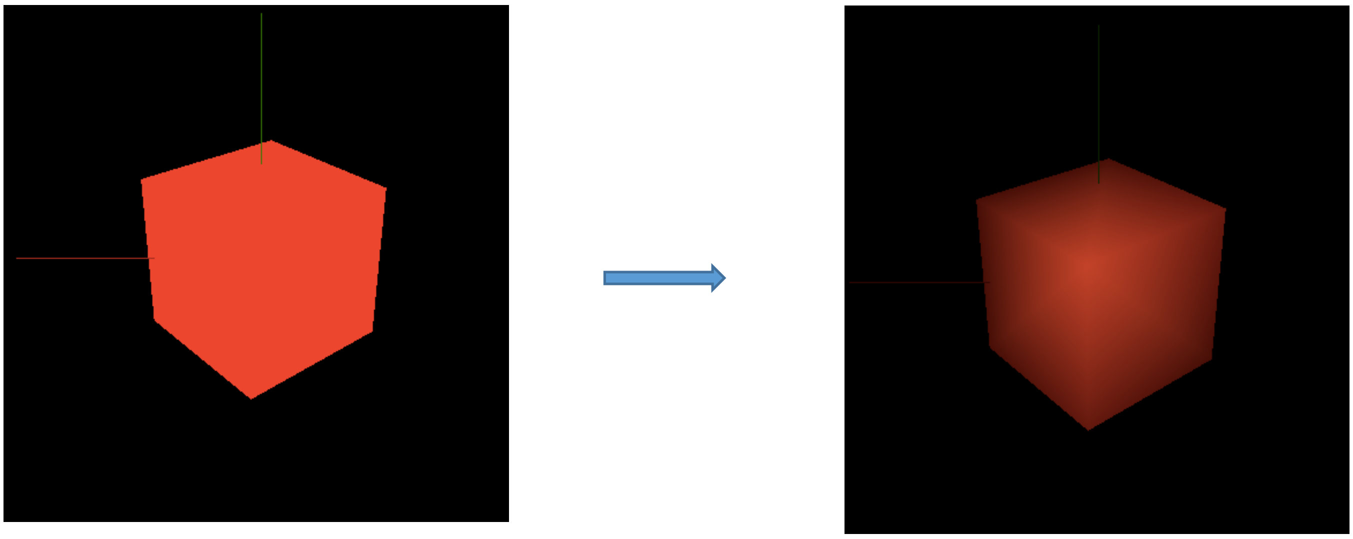
```
/*
 Callucate the cross product of 2 vectors and normalised them
 */
float * calCrossProduct(float u[3], float v[3]){
    static float c[3];
    float element1;
    float element2;
    float element3;
    float magnitude;
    element1 = (u[1]*v[2]-u[2]*v[1]);
    element2 = (u[2]*v[0]-u[0]*v[2]);
    element3 = (u[0]*v[1]-u[1]*v[0]);
    magnitude = sqrtf(powf(element1, 2)+powf(element2, 
2)+powf(element3, 2));
    //Normalize
    c[0] = element1/magnitude;
    c[1] = element2/magnitude;
    c[2] = element3/magnitude;

    return c;
}
```

Creating an array to store the normal for 6 faces:

```
float faceNormal[6][3];// Stores the face normal of each face
```
And calculate the normal for each face:
```
/*
 Get the normal for each face
 */
void assignNormal (float pointA[3],float pointB[3],float
pointC[3],int face){
    float AB[3]; // Creating vectors.
    float AC[3];
    float *temp;
    AB[0] = pointB[0] - pointA[0];
    AB[1] = pointB[1] - pointA[1];
    AB[2] = pointB[2] - pointA[2];


    AC[0] = pointC[0] - pointA[0];
    AC[1] = pointC[1] - pointA[1];
    AC[2] = pointC[2] - pointA[2];

    temp = calCrossProduct(AB, AC);
    faceNormal[face][0] = temp[0];
    faceNormal[face][1] = temp[1];
    faceNormal[face][2] = temp[2];

}
```
After this, we can calculate the normal for each vertex which is the average normal of neighbour faces of the vertex.

Finally, assign the normal to each vertex:
```
/*
 Drawing cube and assign normal to each vertex.
 */
void drawCube(void)
{
    float *average;
    updateNormal();
    glEnable(GL_NORMALIZE);

    for (int i=0; i<6; i++) {
        glBegin(GL_POLYGON);
        glColor3f(1.0f,0.0f,0.0f);
        for (int j =0; j<4; j++) {
            // Calculate average vertex and assign it.
            average=calNormalAverage(faces[verticesForFace[i][j]]);
            glNormal3fv(average);
            glVertex3fv(vertex[verticesForFace[i][j]]);
        }
        glEnd();
    }
}
```
Thus, the Phong illumination has been implemented.

Screenshot of the result:



Requirement 6: Texture mapping

map textures on different faces of the cube

To enable texture mapping, we need to load the texture first and specify parameters to define it. By using tgaload and setuptexture provided by Demo10 code, the loaded texture file brick.tga has now been ready to use.

```
void setuptexture() {
    unsigned char* image = NULL;
    int iheight, iwidth;
    image_t tex_image; //only needed for tga


    // Load image from file
    tgaLoad("brick.tga", &tex_image, TGA_FREE | TGA_LOW_QUALITY);


    // Create a texture object with a unused texture ID
    glGenTextures(1, g_textureID);

    // Set g_textureID as the current 2D texture object
    glBindTexture(GL_TEXTURE_2D, g_textureID[0]);

    // Set the loaded image as the current texture image
    glTexImage2D ( GL_TEXTURE_2D, 0, tex_image.info.tgaColourType,
    tex_image.info.width, tex_image.info.height, 0,
    tex_image.info.tgaColourType, GL_UNSIGNED_BYTE, tex_image.data
);


    // Specify what to do when s, t outside range [0, 1]
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // Specify how to interpolate texture color values
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
}
```

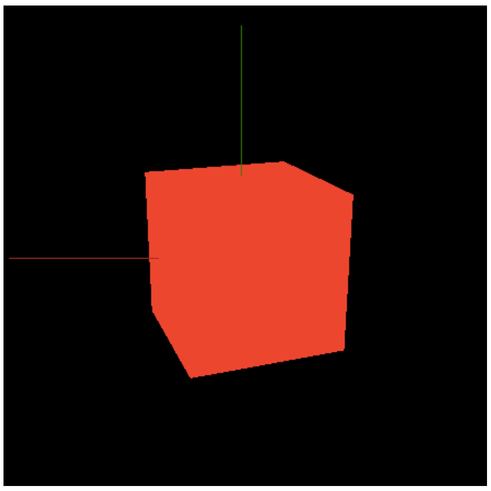
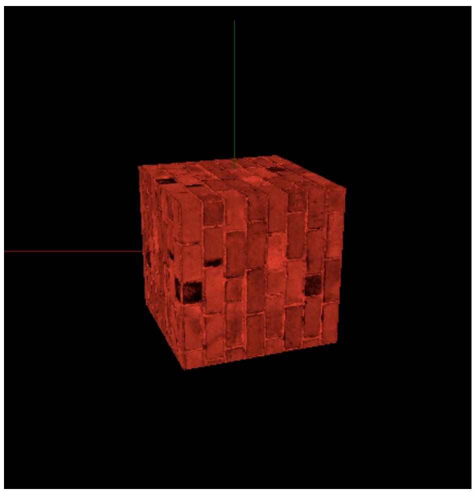
The all-in-one setuptexture() function helped defining a texture by create a texture object and specifying texels etc.
Based on this, all we need is to establish the texture coordinate system:

```
float textureMap[4][2]={{0,0},{0,1},{1,1},{1,0}};
```

OpenGL will place the texture coordinate to the specified image once the glTexCoord2f (x,y) is called. Thus, we can use this to map our texture to our cube:

```
void drawCube(void)
{
    for (int i=0; i<6; i++) {
        glBegin(GL_POLYGON);
        glColor3f(1.0f,0.0f,0.0f);
        for (int j =0; j<4; j++) {
           glTexCoord2fv(textureMap[j]);
           glVertex3fv(vertex[verticesForFace[i][j]]);
        }
        glEnd();
    }
}    //Call glTexCoord2fv when we drawing the cube
```

Screenshot for the result:

CW2 PART

Requirement 6: Load a mesh model

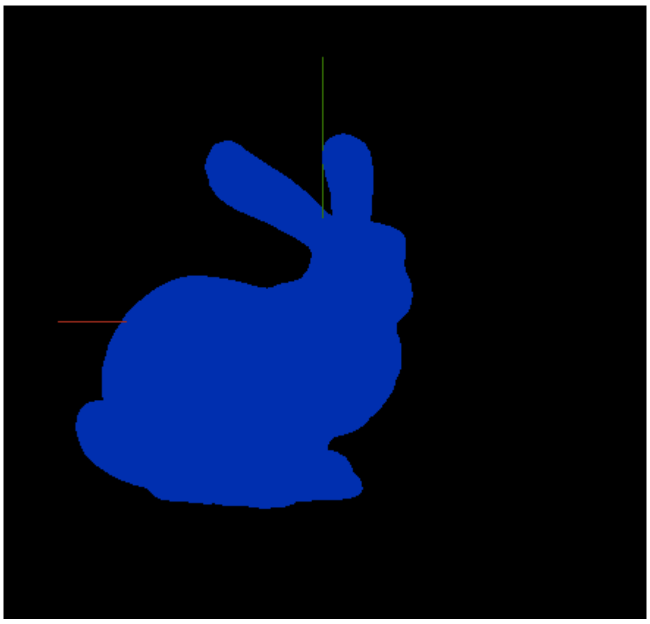
translate and scale the model to fit into the cube
Loading the mesh into our program can be accomplished by objLoader from coursework 2 surpporting code. ObjLoader read the obj file (bunny.obj in my example) and transfer the vertex and face into two Eigen vector :

```
std::vector<Eigen::Vector3d> vertices;
std::vector<Eigen::Vector3d> faceIndices;
```

Next thing is to draw the mesh, the surface of the mesh consists of several triangles, thus we can use GL_TRIANGLES to draw the mesh.
Example Code of drawing bunny:

```
/*
 Drawing the loaded mesh
 */
void drawMesh(void)
{
    for (int i=0;i<faceIndices.size();i++)
    {
        glBegin(GL_TRIANGLES);
        glVertex3f(vertices[faceIndices[i][0]-1][0],
vertices[faceIndices[i][0]-1][1],
                   vertices[faceIndices[i][0]-1][2]);
        glVertex3f(vertices[faceIndices[i][1]-1][0],
vertices[faceIndices[i][1]-1][1],
                   vertices[faceIndices[i][1]-1][2]);
        glVertex3f(vertices[faceIndices[i][2]-1][0],
vertices[faceIndices[i][2]-1][1],
                   vertices[faceIndices[i][2]-1][2]);
        glEnd();
    }
}
```

FaceIndices gives information about the vertex on each face, the index of faceIndices start at number 1, so we need to minus one during the loop.
Screen shot for the result:



In able to fit the cube size, we need to translate the bunny then scale it.
We can form the compound matrix transM by multiplying scale and translate matrix in homogeneous coordinates:

```
// Create the result transformation matrix (translate * scale)
float TransM[4][4] = {{1/scaleVector,0,0,-meshCentreX*(1/scaleVector)},
                      {0,1/scaleVector,0,-meshCentreY*(1/scaleVector)},
                      {0,0,1/scaleVector,-meshCentreZ*(1/scaleVector)},
                      {0,0,0,1}};
```

The scaleVector can be obtained by finding the maximum and minimum value in each vertex's coordinates:
Example code of finding maximum value in x coordinate

```
float meshMaxX()
{
    float maxValue = 0;
    for ( unsigned int i=0; i<vertices.size(); i++ ){
        if(vertices[i][0] >maxValue){
            maxValue = vertices[i][0];
        }
    }
    return maxValue;
}
```

Now we have 3 maximum values and 3 minimum values, finding their sum then divided by 2 is centre coordinates of the mesh:

```
meshCentreX = (maxX + minX)/2;
meshCentreY = (maxY + minY)/2;
meshCentreZ = (maxZ + minZ)/2;
```

The scaleVector is the highest difference between centre and the maximum coordinates in their axis:

```
    scaleX = maxX - meshCentreX;
    scaleY = maxY - meshCentreY;
    scaleZ = maxZ - meshCentreZ;
    // Calculation about scaleVector
    scaleVector = scaleX;
    if(scaleX < scaleY){
        scaleVector = scaleY;
    }
    if (scaleX < scaleZ) {
        scaleVector = scaleZ;
    }
    if (scaleY < scaleZ) {
        scaleVector = scaleZ;
    }
```

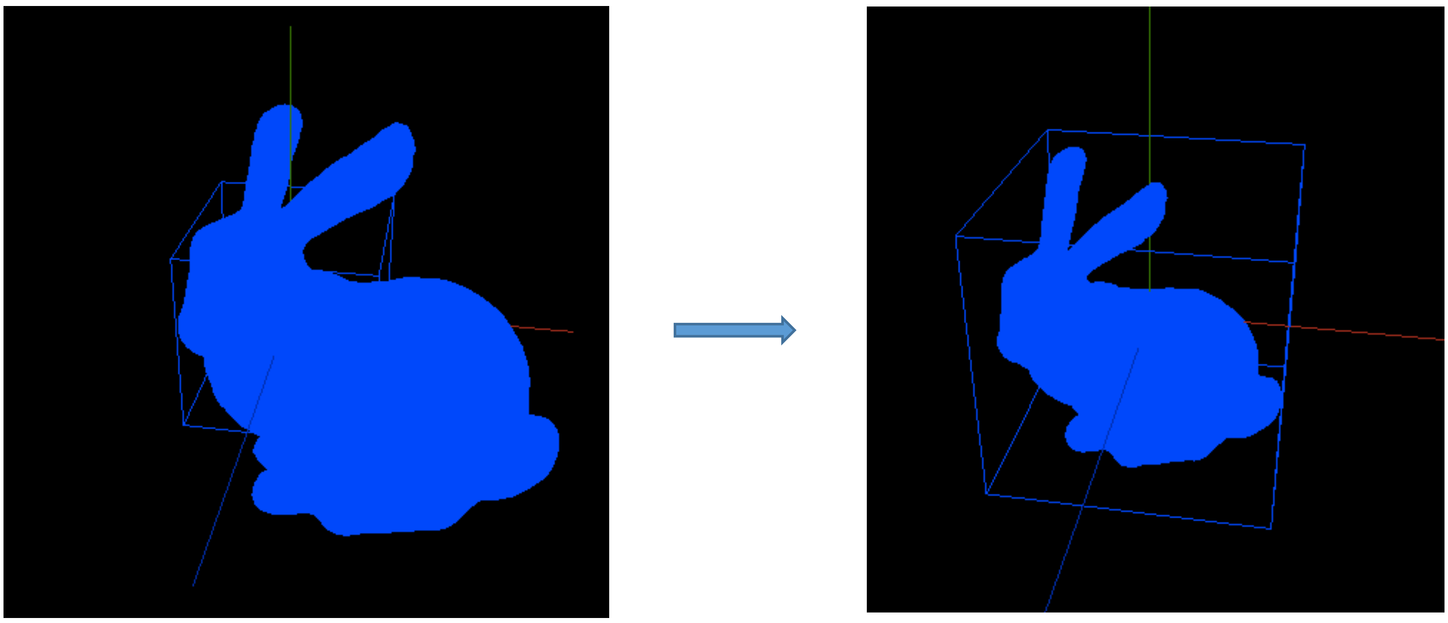
Since the mesh is oversized, we need to scale it down, thus we need the inverse value of scaleVector (1/scaleVector) in our matrix.

Translate vector in this example is the negative value of meshCentre, because we need to shifit to the world space centre by meshCentre amount in x y z directions respectively.

Finally, applying the above matrix to each vertex of the mesh:

```
for (int i=0; i<vertices.size(); i++) {
        float x = vertices[i][0];
        float y = vertices[i][1];
        float z = vertices[i][2];
        vertices[i][0] = (TransM[0][0] * x) + (TransM[0][1] * y) +
(TransM[0][2] * z) + TransM[0][3];
        vertices[i][1] = (TransM[1][0] * x) + (TransM[1][1] * y) +
(TransM[1][2] * z) + TransM[1][3];
        vertices[i][2] = (TransM[2][0] * x) + (TransM[2][1] * y) +
(TransM[2][2] * z) + TransM[2][3];
    }
```

Now, the cube should fit into the cube we have built in CW1.
Screen shot for the result:



Requirement 8: Rotate mesh, render mesh in different mode

as you did for the cube

Three rendering modes

Just as the requirement suggest, rendering mesh is similar to rendering cube, so we can implement these three modes in the same way.

The only difference is that we need to traverse the vertices without create specific array, as the objloader has already done this for us.
Example Code of drawing mesh vertices.

```
/*
 Drawing Vertices of Mesh
 */
void drawMeshVertex(void)
{
    for (int i=0;i<faceIndices.size();i++)
    {
        glBegin(GL_POINTS);
        glVertex3f(vertices[faceIndices[i][0]-1][0],
vertices[faceIndices[i][0]-1][1],
                   vertices[faceIndices[i][0]-1][2]);
        glVertex3f(vertices[faceIndices[i][1]-1][0],
vertices[faceIndices[i][1]-1][1],
                   vertices[faceIndices[i][1]-1][2]);
        glVertex3f(vertices[faceIndices[i][2]-1][0],
vertices[faceIndices[i][2]-1][1],
                   vertices[faceIndices[i][2]-1][2]);
        glEnd();
        glPointSize(1);
    }
}
```

Example Code of drawing mesh edges:

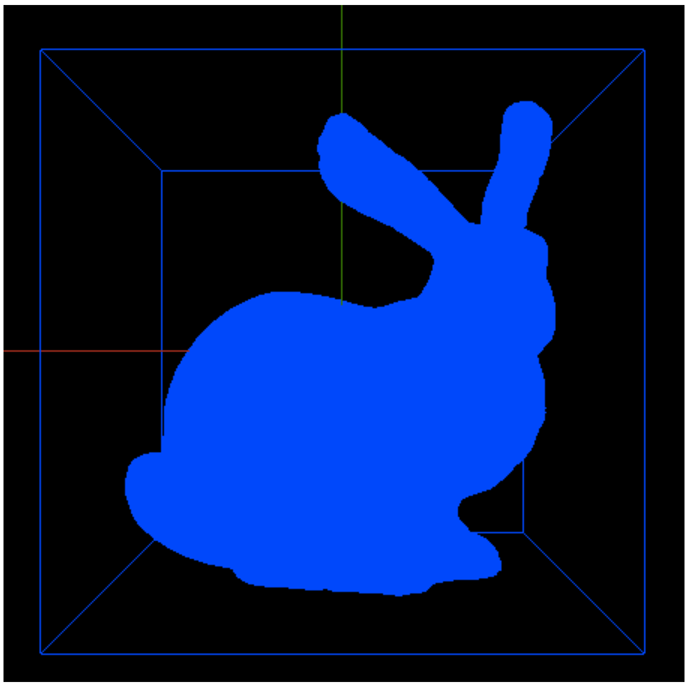
```
/*
 Dawing mesh edges.
 */
void drawMeshEdges(void)
{
    for (int i=0;i<faceIndices.size();i++)
    {
        glBegin(GL_LINES);
//Line one
        glVertex3f(vertices[faceIndices[i][0]-1][0],
vertices[faceIndices[i][0]-1][1],
                   vertices[faceIndices[i][0]-1][2]);
        glVertex3f(vertices[faceIndices[i][1]-1][0],
vertices[faceIndices[i][1]-1][1],
                   vertices[faceIndices[i][1]-1][2]);
//Line two
        glVertex3f(vertices[faceIndices[i][1]-1][0],
vertices[faceIndices[i][1]-1][1],
                   vertices[faceIndices[i][1]-1][2]);
        glVertex3f(vertices[faceIndices[i][2]-1][0],
vertices[faceIndices[i][2]-1][1],
                   vertices[faceIndices[i][2]-1][2]);
//Line three
        glVertex3f(vertices[faceIndices[i][2]-1][0],
vertices[faceIndices[i][2]-1][1],
                   vertices[faceIndices[i][2]-1][2]);
        glVertex3f(vertices[faceIndices[i][0]-1][0],
vertices[faceIndices[i][0]-1][1],
                   vertices[faceIndices[i][0]-1][2]);

        glEnd();
    }
}
```

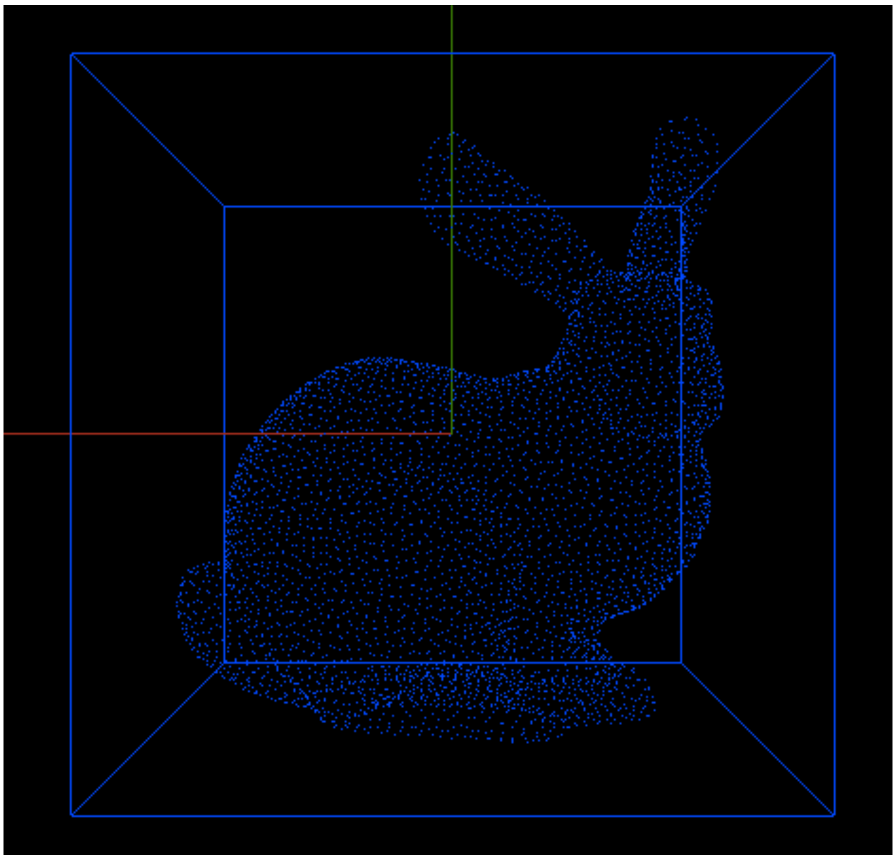
Drawing edges of mesh is slightly different from cube because we don't loop over the specific array to draw lines. The reason is that drawing edges in triangle is much simpler than in cube, and the index of vertex is fixed this time (always [0][1][2]).

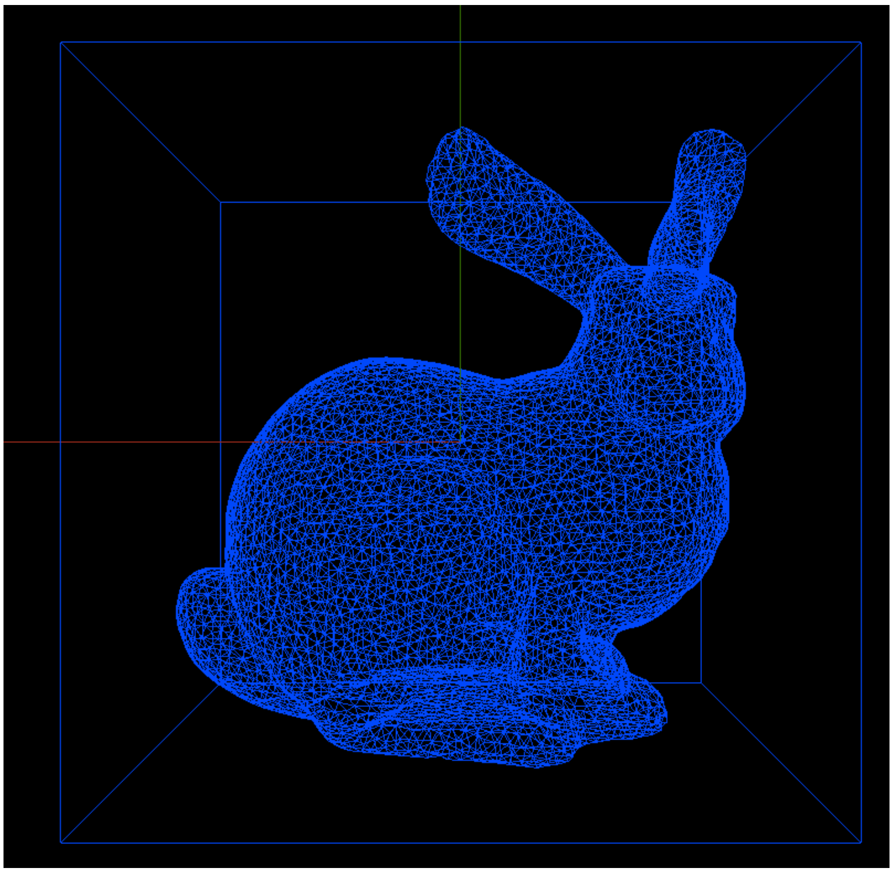
Assign the key callback function:

```
void display (void) {
    . . .
    // different render mode*/
    switch (rendermode ) {
        case 'f': // to display faces
            drawMesh();
            break;
        case 'v': // to display points
            drawMeshVertex();
            break;
        case 'e': // to display edges
            drawMeshEdges();
            break;

    }. . .
}
void keyboard ( unsigned char key, int x, int y )
{   . . .
     // Switch render mode for v,e,f
     case 'v':
         rendermode='v';
         break;
     case 'e':
         rendermode='e';
         break;

     case 'f':
         rendermode='f';
         break;
    . . .
}
```

Screenshot for the result:







Mesh rotation

Rotate the mesh also quite similar to rotate the cube. We can use the same rotaion matrix Rx,Ry,Rz to rotate out mesh.

Recall: Rotation Matrix Rx:

```
float Rx[4][4] = {{1,0,0,0},
                 {0, cosf(-meshAngleX),-sinf(-meshAngleX),0},
                 {0,sinf(-meshAngleX),cosf(-meshAngleX),0},
                 {0,0,0,1}};
```

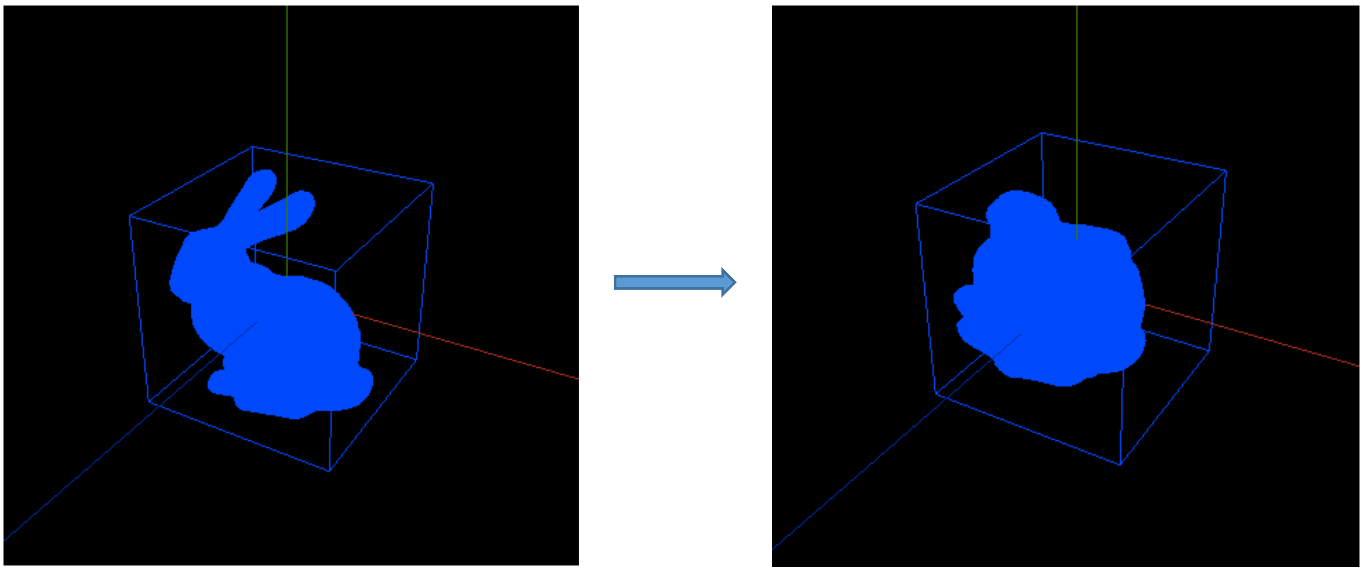
Example Code of applying rotaion matrix Rx to vertices:

```
//Applying Rx matrix to every points in mesh vertices.
    for (int i=0; i<vertices.size(); i++) {
        float x = vertices[i][0];
        float y = vertices[i][1];
        float z = vertices[i][2];
        vertices[i][0] = (Rx[0][0] * x) + (Rx[0][1] * y) + (Rx[0][2]
           * z) + Rx[0][3];
        vertices[i][1] = (Rx[1][0] * x) + (Rx[1][1] * y) + (Rx[1][2]
           * z) + Rx[1][3];
        vertices[i][2] = (Rx[2][0] * x) + (Rx[2][1] * y) + (Rx[2][2]
           * z) + Rx[2][3];
    }
```

Assigning key callback function:

```
void keyboard ( unsigned char key, int x, int y )
{   . . .
        // Mesh operation
        case 's':
            rotateMeshX();
            break;
     . . .
}
```

Screenshot of the result:



Shading effect

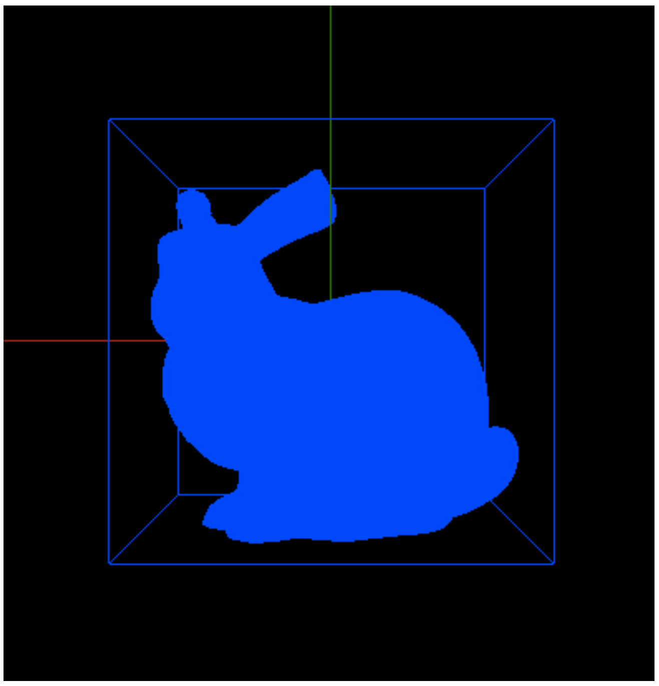
Add shading effect to mesh needs to assign normal to each vertex just as what we do for the cube. So the process is same:
Example code of calculating the normal:

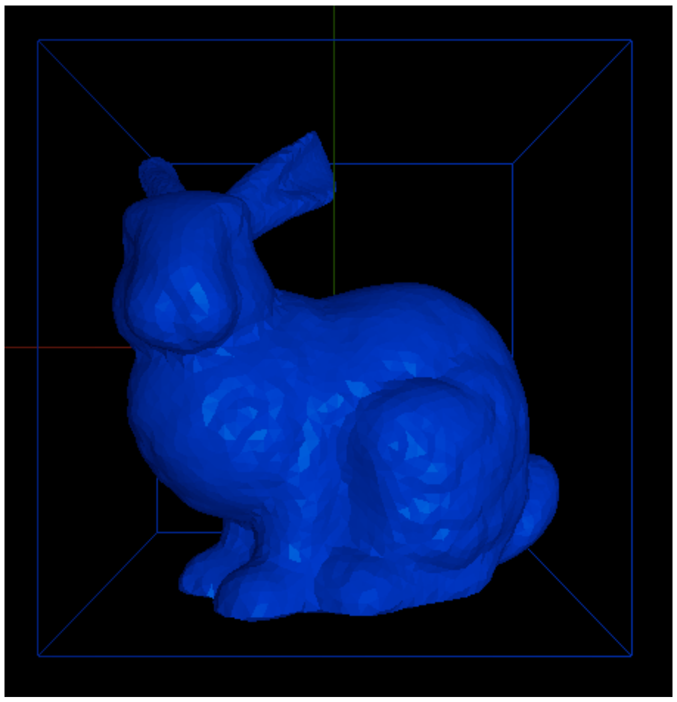
```
void drawMesh(void)
{ ...

//Create two vectors.
float AB[3];
float AC[3];
float *temp;

AB[0] = vertices[faceIndices[i][1]-1][0] - vertices[faceIndices[i][0]-1][0];
AB[1] = vertices[faceIndices[i][1]-1][1] - vertices[faceIndices[i][0]-1][1];
AB[2] = vertices[faceIndices[i][1]-1][2] - vertices[faceIndices[i][0]-1][2];

AC[0] = vertices[faceIndices[i][2]-1][0] - vertices[faceIndices[i][0]-1][0];
AC[1] = vertices[faceIndices[i][2]-1][1] - vertices[faceIndices[i][0]-1][1];
AC[2] = vertices[faceIndices[i][2]-1][2] - vertices[faceIndices[i][0]-1][2];

temp = calCrossProduct(AB, AC);
```

After that, assign the normal to each vertex:

```
 glNormal3fv(temp);
 glVertex3f(vertices[faceIndices[i][0]-1][0], vertices[faceIndices[i][0]-1][1],
            vertices[faceIndices[i][0]-1][2]);
 glNormal3fv(temp);
 glVertex3f(vertices[faceIndices[i][1]-1][0], vertices[faceIndices[i][1]-1][1],
            vertices[faceIndices[i][1]-1][2]);
 glNormal3fv(temp);
 glVertex3f(vertices[faceIndices[i][2]-1][0], vertices[faceIndices[i][2]-1][1],
            vertices[faceIndices[i][2]-1][2]);

... }
```

By keeping the same light and material, the shading effect has now been implemented.
Screenshot of the result:





Requirement 9: Compute axis-aligned bounding box (abb) of the mesh

Use transparent material to render abb

Abb is a cuboid that fixed at each axis (axis-aligned). So drawing the abb is similar to drawing the cube. But the vertex of the abb is not constant, they may vary as we rotate the bunny.

The vertex of the abb is formed by maximum and minimum value in x,y,z. Thus, all we need to do is to work out them first.

Example Code of calculating maximum value in x axis:

```
float meshMaxX()
{
    float maxValue = 0;
    for ( unsigned int i=0; i<vertices.size(); i++ ){
        if(vertices[i][0] >maxValue){
            maxValue = vertices[i][0];
        }
    }
    return maxValue;
}
```

After that, use these value to form the vertex of abb:

```
float vertexMesh[8][3] =
{{maxX,maxY,maxZ},{minX,maxY,maxZ},{minX,minY,maxZ},{maxX,minY,maxZ}
,{maxX,minY,minZ},{minX,minY,minZ},{maxX,maxY,minZ},{minX,maxY,minZ}
};
```

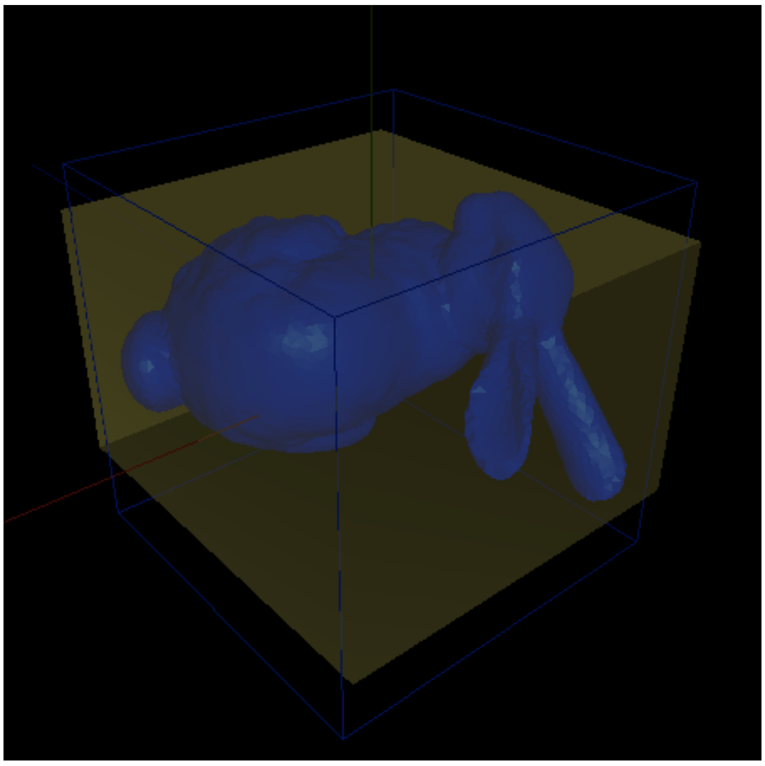
And use these vertices to draw the abb just as drawing the cube in Req 1, since we need to use transparent material, we also need to blend the abb color with canvas to create the transparent effect:

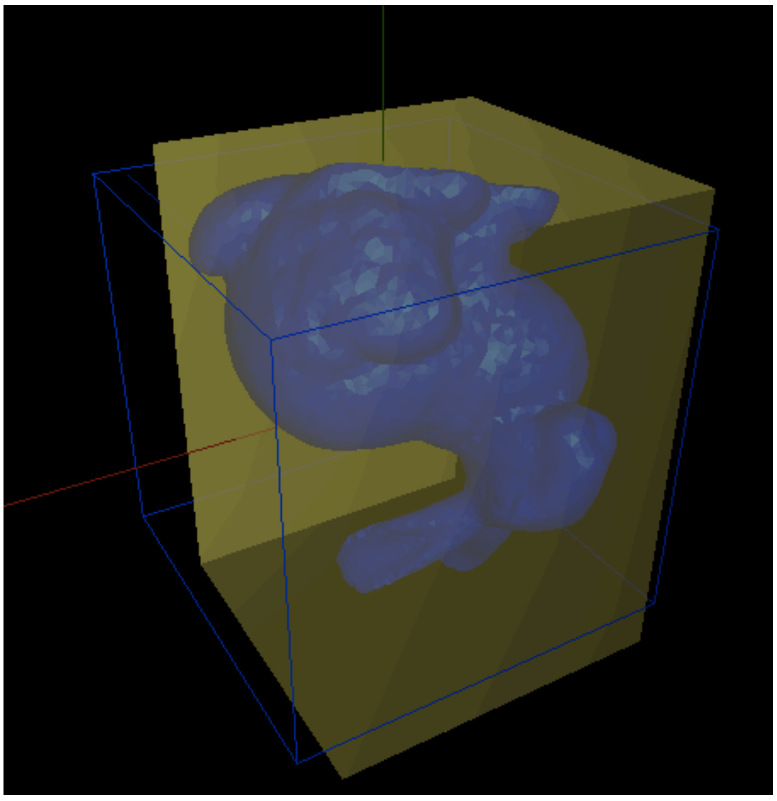
Example Code of drawing a transparent abb.

```
void drawAbb(){
    float vertexMesh[8][3] =
{{maxX,maxY,maxZ},{minX,maxY,maxZ},{minX,minY,maxZ},{maxX,minY,maxZ}
,{maxX,minY,minZ},{minX,minY,minZ},{maxX,maxY,minZ},{minX,maxY,minZ}
};
    glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    for (int i=0; i<6; i++) {
        glBegin(GL_POLYGON);
        glColor4f(1.0f, 1.0f, 0.4f, 0.5f);
        for (int j =0; j<4; j++) {
            glVertex3fv(vertexMesh[verticesForFace[i][j]]);
        }
        glEnd();
    }
}
```

The maximum and minimum will change as the bunny rotate, thus we need to put our max/min calculation function and drawAbb() into display().

Screenshot for the result:





Requirement 10: Compute oriented bounding box (obb) of the mesh

The obb fixed tightly with the mesh and will rotate with the mesh without changing the size. According to this, we can build the obb by working out the vertex and applying the same rotation matrix to them.

First of all, we need to work out the covariance and mean of vertices. Let's strat with the mean first.

```
/*
 Calculate the mean of specified coordinates
 */

float calMean(int index){ // 0 for x, 1 for y , 2 for z
    float mean = 0;
    for (int i=0; i<vertices.size(); i++) {
        mean += vertices[i][index];
    }
    mean = mean/vertices.size();
    return mean;
}
```

The above function will return the mean for the given index.

Now we need to compute the covariance of the input vertices and form the covariance matrix.

According to covariance equation : $\frac{1}{n}\sum_{i=1}^{n}(a_{ij}-\mu_j)(a_{ik}-\mu_k)$

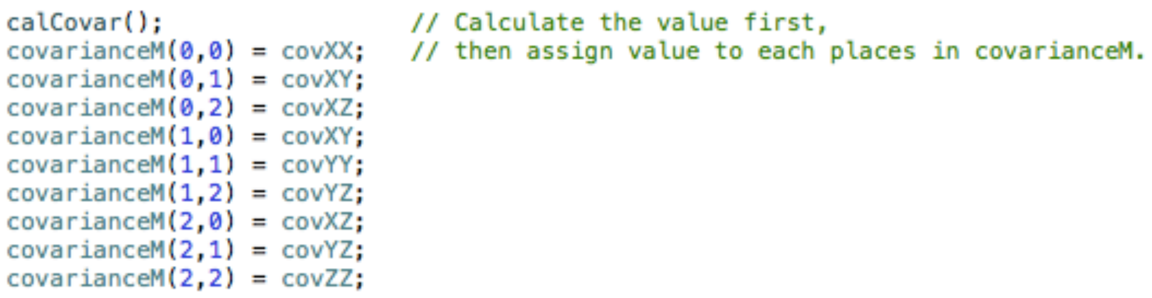
We can build our program to calculate the covariance as follow:

```
/*
 Calculate the covariance of vertices in mesh
 */
void calCovar(){
    for (int i ; i<vertices.size(); i++) {
        covXX += (vertices[i][0]-calMean(0))*(vertices[i][0]-
calMean(0));
        covXY += (vertices[i][0]-calMean(0))*(vertices[i][1]-
calMean(1));
        covXZ += (vertices[i][0]-calMean(0))*(vertices[i][2]-
calMean(2));
        covYY += (vertices[i][1]-calMean(1))*(vertices[i][1]-
calMean(2));
        covYZ += (vertices[i][1]-calMean(1))*(vertices[i][2]-
calMean(2));
        covZZ += (vertices[i][2]-calMean(2))*(vertices[i][2]-
calMean(2));

    }
    covXX = covXX/vertices.size();
    covXY = covXY/vertices.size();
    covXZ = covXZ/vertices.size();
    covYY = covYY/vertices.size();
    covYZ = covYZ/vertices.size();
    covZZ = covZZ/vertices.size();
}
```

And form the covariance matrix:

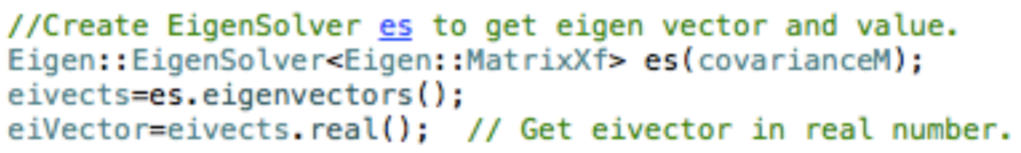
```
Eigen::Matrix3f covarianceM;
```

```
Assign Value:
```



```
calCovar();                  // Calculate the value first,
covarianceM(0,0) = covXX;    // then assign value to each places in covarianceM.
covarianceM(0,1) = covXY;
covarianceM(0,2) = covXZ;
covarianceM(1,0) = covXY;
covarianceM(1,1) = covYY;
covarianceM(1,2) = covYZ;
covarianceM(2,0) = covXZ;
covarianceM(2,1) = covYZ;
covarianceM(2,2) = covZZ;
```

Now, we need the Eigen vector according the covariance matrix and the transpose of Eigen vector.



```
//Create EigenSolver es to get eigen vector and value.
Eigen::EigenSolver<Eigen::MatrixXf> es(covarianceM);
eivects=es.eigenvectors();
eiVector=eivects.real();  // Get eivector in real number.
```

By applying the equation: $\boldsymbol{p'} = \boldsymbol{U^T}(\boldsymbol{p} - \boldsymbol{\mu})$

We can transfer our vertices into new object reference frame because Eigen vector provide a reference frame that vertices are clustered.

Example Code of computing the maximum and minimum value of vertices in new object reference frame:

```
for (int i=0; i<vertices.size(); i++) {

        tempx = eiVector.transpose()(0,0)*(vertices[i][0]-meanx) + 
eiVector.transpose()(0,1)*(vertices[i][1]-meany)+
eiVector.transpose()(0,2)*(vertices[i][2]-meanz);
        if (obbMaxX < tempx ) {
            obbMaxX = tempx;
        }
        if (obbMinX > tempx ) {
            obbMinX = tempx;
        }
```

Using the same way, we can get 3 maximum and 3 minimum value of x,y,z in new reference frame. They are the coordinate of our obb just like abb and add values to obbPoints matrix:

Example Code of adding the first point of obb into obbPoints matrix:

```
    obbPoints[0][0] = obbMaxX;
    obbPoints[0][1] = obbMaxY;
    obbPoints[0][2] = obbMaxZ;
```

Now all we need is to transfer our max and min value back to the world space in order to plot them to the canvas. The equation is $\boldsymbol{P} = (\boldsymbol{V}^{\boldsymbol{T}})^{-\boldsymbol{1}}\boldsymbol{P}' + \boldsymbol{M}$

```
/*
 Work out the coordinates of obb points.
 */
void calOBBPoints(){
    eiVectorInverse = eiVector.transpose().inverse();
    for (int i =0; i<8; i++) {
        float x, y, z;

        x =obbPoints[i][0];
        y =obbPoints[i][1];
        z =obbPoints[i][2];

        obbPoints[i][0] = eiVectorInverse(0,0)*(x) +
eiVectorInverse(0,1)*(y)+ eiVectorInverse(0,2)*(z) +calMean(0);
        obbPoints[i][1] = eiVectorInverse(1,0)*(x) +
eiVectorInverse(1,1)*(y)+ eiVectorInverse(1,2)*(z) +calMean(1);
        obbPoints[i][2] = eiVectorInverse(2,0)*(x) +
eiVectorInverse(2,1)*(y)+ eiVectorInverse(2,2)*(z) +calMean(2);
    }
}
```

The final thing is to draw the obb as we have calculated all the vertices of our obb:

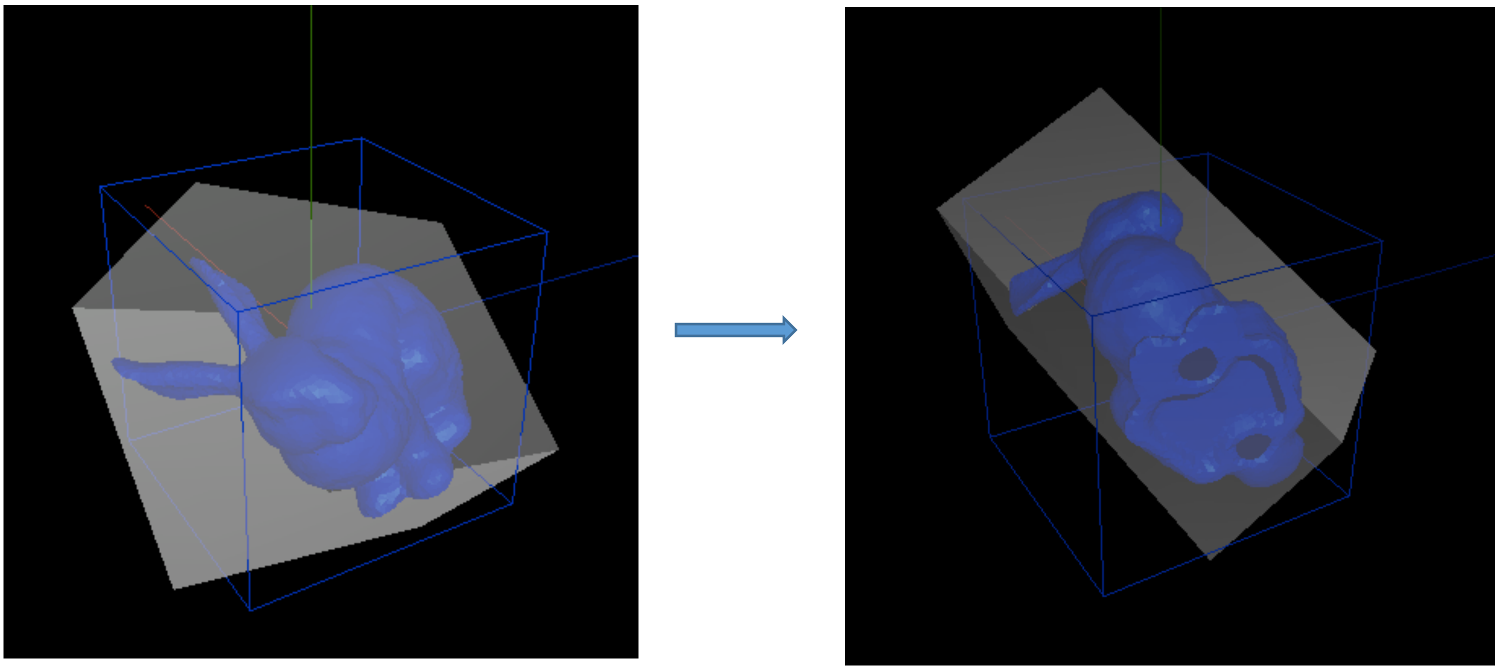
```
void drawOBB(){
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    for (int i=0; i<6; i++) {
        glBegin(GL_QUADS);
        glColor4f(1.0f, 1.0f, 1.0f, 0.5f);
        for (int j =0; j<4; j++) {
            glVertex3fv(obbPoints[verticesForFace[i][j]]);
        }
        glEnd();
    }
}
```

Since the obb should rotate with our bunny, so in the rotation matrix, we should also rotate the obb vertices as well.

Example Code of obb vertices about X-axis rotation:

```
void rotateMeshX(){ . . .
//Applying Rx matrix to every points in obb vertices.
    for (int i=0; i<8 ; i++) {
        float x = obbPoints[i][0];
        float y = obbPoints[i][1];
        float z = obbPoints[i][2];
        obbPoints[i][0] = (Rx[0][0] * x) + (Rx[0][1] * y) +
(Rx[0][2] * z) + Rx[0][3];
        obbPoints[i][1] = (Rx[1][0] * x) + (Rx[1][1] * y) +
(Rx[1][2] * z) + Rx[1][3];
        obbPoints[i][2] = (Rx[2][0] * x) + (Rx[2][1] * y) +
(Rx[2][2] * z) + Rx[2][3];
. . .}
```

Screenshot for the result:



Shortcomings and future plans of the program

The biggest problem of this program is that it does not implement the final requirement which is the clip plane.

The rotation action is based on Euler angles, thus the disadvantage is obvious, the rotation object could suffer Gimbal Lock at some point.

The interaction between panning and rotating the camera is not smooth, there will be a bad observation as the object will get little bigger.

Abb and Obb has not been assigned normal, there could be a reflection at certain point.

If there is no time limit, I could spend as much time as possible to implement the final requirement because the large amount of computation of clip plane do takes time.

The Euler angle can be replaced by Quaternion. Quaternion has better performance and no gimbal lock.

To improve the Abb and Obb, the best way is to calculate the normal for them.